CEWES Major Shared Resource Center

MSRC

CEWES MSRC/PET TR/99-06

PAPI: Portable Interface to Hardware Performance Counters

by

Shirley Browne
George Ho
Phil Mucci

04h00499

Work funded by the DoD High Performance Computing Modernization Program CEWES Major Shared Resource Center through

Programming Environment and Training (PET)

Supported by Contract Number: DAHC94-96-C0002
Nichols Research Corporation

Views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of Defense Position, policy, or decision unless so designated by other official documentation.

PAPI: Portable Interface to Hardware Performance Counters

Shirley Browne
George Ho
Phil Mucci

University of Tennessee
Innovative Computing Laboratory

Overview

The purpose of the **PAPI** project is to specify a standard application programming interface (API) for accessing hardware performance counters available on most modern microprocessors. These counters exist as a small set of registers that count *Events*, occurrences of specific signals related to the processor's function. Monitoring these events facilitates correlation between the structure of source/object code and the efficiency of the mapping of that code to the underlying architecture. This correlation has a variety of uses in performance analysis including hand tuning, compiler optimization, debugging, benchmarking, monitoring and performance modeling. In addition, it is hoped that this information will prove useful in the development of new compilation technology as well as in steering architectural development towards alleviating commonly occurring bottlenecks in high performance computing.

Description

The **PAPI** provides two interfaces to the underlying counter hardware; a simple, high level interface for the acquisition of simple measurements and a fully programmable, low level interface directed towards users with more sophisticated needs. The low level **PAPI** deals with hardware events in groups called *EventSets*. EventSets can reflect how the counters are most frequently used, such as taking simultaneous measurements of different hardware events and relating them to one another. For example, relating cycles to memory references or flops to level 1 cache misses can help detect poor locality and memory management. In addition, EventSets allow a highly efficient implementation which translates to more detailed and accurate measurements. EventSets are fully programmable and have features such as guaranteed thread safety, writing of counter values, multiplexing and notification on threshold crossing, as well as processor-specific features. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.

PAPI provides portability across different platforms. It uses the same routines with similar argument lists to control and access the counters for every architecture. As part of **PAPI,** we have predefined a set of events that we feel represents the lowest common denominator of every *good* counter implementation. Our intent is that the same source code will count *similar and possibly comparable* events when run on different platforms. If the programmer chooses to use this set of standardized events, then the source code need not be changed and only a fresh compilation and link is necessary. However, should the developer wish to access machine-specific events, the low level API provides access to all available events and counting modes. If an event or feature does not exist on the current platform, **PAPI** returns an appropriate error code. This significantly reduces the porting effort of code using the **PAPI** because the form of each call to **PAPI** remains the same, just the argument lists need updating in the case of machine-specific events. In addition to the standard set, each **PAPI** supports *all native events* through the ability to directly accept platform specific counter numbers. Definitions for most, if not all of these, are included as conditional macros in the PAPI header file. In this way, **PAPI** avoids the necessity for inefficient code to translate all events for all platforms into a uniform representation and back again. This translation is only done for the relatively few events defined in the standardized set.

Some processors like those in the **Cray Vector** and the **IBM POWER** series have counter groups. They enable access to specific groups of counters, instead of individual events. This presents a serious portability problem, thus **PAPI** abstracts hardware counters from their groups with a packed naming scheme. Each counter control value or event is made up of the counter group number and the number of the specific counter in that group.

PAPI reference implementation can be divided into two layers of software. The upper layer consists of the API and machine independent support functions. The lower layer defines and exports a machine independent interface to machine dependent functions and data structures. These functions access the *substrate*, which may consist of the operating system, a kernel extension or assembly functions to directly access the processor's registers. The **PAPI** reference implementation tries to use the most efficient and flexible of the three, depending on what is available. Naturally, the functionality of the upper layers heavily depends on that provided by the substrate. In cases where the substrates do not provide highly desirable features, **PAPI** attempts to emulate them as described below. However, one particular difficulty **PAPI** cannot solve is the issue of thread/processor affinity. Event though the API is thread safe, it cannot guarantee that the same processor runs the same thread on every context switch. Thus for implementations of **PAPI** without per thread counter functionality in the kernel, some anomalies may result. This rarely happens in practice, as operating system schedulers give preference to the previous processor in order to minimize cache thrashing and bus traffic.

The **PerfAPI** has the capability to internally multiplex hardware events if the operating system or counter interface does not support it. This functionality presents the developer with the view that *all events are countable all the time.* Naturally, the multiplexing of counter events incurs a small amount of overhead and can adversely affect the accuracy of reported counter values. Nevertheless, similar features have proved quite successful in commercial implementations of counter software found in **SGI's IRIX 6.x** and **DEC's Digital Unix v4.x.** Multiplexing has been shown to be especially useful in the performance tuning process as a means of looking for outstanding bottlenecks when first analyzing an unfamiliar code. When multiplexing is enabled and subsequently activated through the request of a conflicting event, the user is informed through a specific success code. In this way, the user is cautioned in deriving any performance data that may not be entirely accurate.

PAPI guards against overflow of counter values. Each counter can potentially be incremented multiple times in a single clock cycle. This combined with increasing clock speeds and the small precision of some of the physical counters means that overflow is likely to occur on platforms where 64-bit counters are not supported in hardware or by the operating system. In those cases, **PAPI** implementations are required to implement 64-bit counters in software. PAPI also provides for asynchronous notification when counters exceed some *user specified* value. Doing so allows the generation of a histogram of the frequency of overflows for a region of code. This functionality provides the basis for all source level performance analysis software, from the antiquated days of **AT & T's prof** to **SGI's SpeedShop.** *Thus for any architecture with even the most rudimentary access to hardware performance counters,* ***PAPI***

provides the foundation for truly portable, source level, performance analysis tools based on real processor statistics.

Standardized Event Definitions

The following is a table of hardware events deemed relevant and useful in tuning application performance. These events have identical assignments in the header files on different platforms *however they may differ in their actual semantics. In addition, all of these events are not guaranteed to be present on all platforms. Please check your platform's documentation carefully.*

Value	**Symbol**	**Description**
`0x80010000`	`PAPI_L1_DCM`	Level 1 data cache misses
`0x80010001`	`PAPI_L1_ICM`	Level 1 instruction cache misses
`0x80010002`	`PAPI_L2_DCM`	Level 2 data cache misses
`0x80010003`	`PAPI_L2_ICM`	Level 2 instruction cache misses
`0x80010004`	`PAPI_L3_DCM`	Level 3 data cache misses
`0x80010005`	`PAPI_L3_ICM`	Level 3 instruction cache misses
`0x80011000`	`PAPI_CA_SHR`	Request for access to shared cache line (SMP)
`0x80011001`	`PAPI_CA_CLN`	Request for access to clean cache line (SMP)
`0x80011002`	`PAPI_CA_INV`	Cache line invalidation (SMP)
`0x80020000`	`PAPI_TLB_DM`	Data translation lookaside buffer misses
`0x80020001`	`PAPI_TLB_IM`	Instruction translation lookaside buffer misses
`0x80021000`	`PAPI_TLB_SD`	Translation lookaside buffer shootdowns (SMP)
`0x80030000`	`PAPI_BRI_MSP`	Branch instructions mispredicted
`0x80030001`	`PAPI_BRI_TKN`	Branch instructions taken
`0x80030002`	`PAPI_BRI_NTK`	Branch instructions not taken
`0x80100000`	`PAPI_TOT_INS`	Total instructions executed
`0x80100001`	`PAPI_INT_INS`	Integer instructions executed
`0x80100002`	`PAPI_FP_INS`	Floating point instructions executed
`0x80100003`	`PAPI_LD_INS`	Load instructions executed
`0x80100004`	`PAPI_SR_INS`	Store instructions executed
`0x80100005`	`PAPI_CND_INS`	Branch instructions executed
0`x80100006`	`PAPI_VEC_INS`	Vector/SIMD instructions executed
`0x80100007`	`PAPI_FLOPS`	Floating Point Instructions executed per second
`0x80200000`	`PAPI_TOT_CYC`	Total cycles
`0x80200001`	`PAPI_MIPS`	Millions of instructions executed per second

Return Codes

All of the functions contained in the **PerfAPI** return standardized error codes. Values greater than or equal to zero indicate success, less than zero indicates failure.

Value	Symbol	Definition
`1`	`PAPI_OK_MPX`	No error, multiplexing has been enabled and is now active
`0`	`PAPI_OK`	No error
`-1`	`PAPI_EINVAL`	Invalid argument
`-2`	`PAPI_ENOMEM`	Insufficient memory
`-3`	`PAPI_ESYS`	A System or C library call failed, please check `errno`
`-4`	`PAPI_ESBSTR`	Substrate returned an error, usually the result of an unimplemented feature
`-5`	`PAPI_ECLOST`	Access to the counters was lost or interrupted
`-6`	`PAPI_EBUG`	Internal error, please send mail to the developers
`-7`	`PAPI_ENOEVNT`	Hardware Event does not exist
`-8`	`PAPI_ECNFLCT`	Hardware Event exists, but cannot be counted due to counter resource limitations
`-9`	`PAPI_ENOTRUN`	No Events or EventSets are currently counting

Constants

The following constants are defined in the header files, **`papi.h`** and **`papi.inc`**.

Value	Symbol	Description
`-1`	`PAPI_NULL`	A nonexistent hardware event used as a place holder
`0`	`PAPI_USER`	Counts are accumulated for events occurring in the user context
`1`	`PAPI_KERNEL`	Counts are accumulated for events occurring in the kernel context
`2`	`PAPI_SYSTEM`	Counts are accumulated for events in all contexts
`0`	`PAPI_PER_THR`	Counts are accumulated on a per kernel thread basis
`1`	`PAPI_PER_PROC`	Counts are accumulated on a per process basis
`2`	`PAPI_PER_CPU`	Counts are accumulated on a per CPU basis
`3`	`PAPI_PER_NODE`	Counts are accumulated on a per node basis
`1`	`PAPI_RUNNING`	EventSet is running
`2`	`PAPI_STOPPED`	EventSet is stopped
`0`	`PAPI_QUIET`	Option to not do any automatic error reporting to `stderr`
`1`	`PAPI_VERB_ECONT`	Option to automatically report any return codes < 0 to `stderr`
`2`	`PAPI_VERB_ESTOP`	Option to automatically report any return codes < 0 to `stderr` and call `exit(PAPI_ERROR)`
`1`	`PAPI_SET_MPXRES`	Option to enable and set the resolution of the multiplexing software
`2`	`PAPI_GET_MPXRES`	Option to query the status of the multiplexing software
`1000`	`PAPI_DEF_MPXRES`	Default resolution in microseconds of the multiplexing software
`3`	`PAPI_DEBUG`	Option to turn on debugging features of the **PerfAPI** library
`4`	`PAPI_SET_OVRFLO`	Option to turn on the overflow reporting software
`5`	`PAPI_GET_OVRFLO`	Option to query the status of the overflow reporting software
`1`	`PAPI_ONESHOT`	Option to have the overflow handler called once
`2`	`PAPI_RANDOMIZE`	Option to have the threshold of the overflow handler randomized
`16`	`PAPI_MAX_EVNTS`	The maximum number of simultaneous events countable by the platform specific hardware *without multiplexing*
`123`	`PAPI_ERROR`	Exit code for **PerfAPI** executables that have `PAPI_VERB_ESTOP` option set

The Low Level API

The following functions represent the low level portion of the PerfAPI. These functions provide greatly increased efficiency and functionality over the high level API presented in the next section. All of the following functions are callable from both C and Fortran except where noted. As mentioned in the introduction, the low level API is only as powerful as the substrate upon which it is built. Thus some features may not be available on every platform. The converse may also be true, that more advanced features may be available and defined in the portion of the header file for a particular platform. The user is encouraged to read platform documentation carefully to determine what features are available.

```
int PAPI_set_granularity(int granularity)
```

This function sets the *measurement granularity* in which the counters function. Here `granularity` is one of the constants, `PAPI_PER_THR, PAPI_PER_PROC, PAPI_PER_CPU` or `PAPI_PER_NODE` as defined in the header file. These constants correspond to their descriptions in the above table. By default, the granularity is set to the most restrictive supported by the substrate.

```
int PAPI_set_context(int context)
```

This function sets the *execution context* in which events are counted. Here `context` is one of the constants `PAPI_USER, PAPI_KERNEL, PAPI_SYSTEM` as defined in the header file.

```
int PAPI_perror(int code, char *destination, int length)
```

This function copies `length` number of characters from the error description string corresponding to `code` into `destination.` The resulting string is always null terminated. If `length` is 0, then the string is printed on `stderr` instead.

```
int PAPI_add_event(int *EventSet, int Event)
```

This function sets up a new EventSet *or* modifies an existing one. To create a new EventSet, `EventSet` must be set to `PAPI_NULL`. Separate EventSets containing events that require use of the same hardware may exist, but an EventSet may not be started if a conflicting EventSet is running. Returns `PAPI_ENOEVNT` if `Event` cannot be counted on this platform. The addition of a conflicting event to an event set will return an error unless `PAPI_SET_MPXRES` has been set. Note: EventSet 0 may not be used; it has been reserved for internal use.

```
int PAPI_add_events(int *EventSet, int *Events, int number)
```

Same as above for a vector of events. If one or more of `Events` cannot be counted on this platform, then this call fails and `PAPI_ENOEVNT` is returned. In addition, the invalid entries in the `Events` array are set to `PAPI_NULL` such that the user can successfully reissue the call.

```
int PAPI_add_pevent(int *EventSet, int code, void *inout)
```

This function allocates a new EventSet for a native programmable Event. Such EventSets can only consist of one event, namely that which is specified in this call. Its semantics are very similar to that of `ioctl()` system call. `inout` points to an opaque data structure that is specific to the value in `code`. Higher level macros may be provided in the header file. Please check the documentation for each substrate. This function has a C binding only.

```
int PAPI_rem_event(int EventSet, int Event)
```

This function removes the hardware counter `Event` from `EventSet`.

```
int PAPI_rem_events(int EventSet, int *Events, int number)
```

This function performs the same as above except for a vector of hardware Events.

`int PAPI_list_events(int EventSet, int *Events, int *number)`

This function decomposes `EventSet` into the hardware Events its contains. `number` is both an input and output parameter.

`int PAPI_start(int EventSet)`

This function starts counting all the hardware Events contained in `EventSet`. All counts are implicitly initialized to zero. As mentioned before, separate EventSets containing events that require use of the same hardware may exist, but may not be started if a conflicting EventSet is running.

`int PAPI_stop(int EventSet, long long *values)`

This function terminates the counting of all hardware Events contained in `EventSet`. In addition, the counters contained in that EventSet are copied into the `values` array.

`int PAPI_read(int EventSet, long long *values)`

This function copies the running or stopped counters in `EventSet` into the `values` array. Internal counters will *not* be re-initialized to zero.

`int PAPI_accum(int EventSet, long long *values)`

This function accumulates the running or stopped counters in `EventSet` into the values array. In addition, it initializes the internal counters to zero.

`int PAPI_write(int EventSet, long long *values)`

This function assigns the values contained in `values` to the internal counters of the Events contained in `EventSet`.

`int PAPI_reset(int EventSet)`

This function initializes the internal counters of the hardware Events contained in `EventSet` to zero.

`int PAPI_cleanup(int *EventSet)`

This function effectively removes `EventSet` from existence. The EventSet must be stopped in order for this call to succeed.

`int PAPI_state(int EventSet, int *status)`

This function returns the state of the *entire* EventSet in `status`. If the call succeeds, then status is either `PAPI_RUNNING` or `PAPI_STOPPED.`

`int PAPI_set_opt(int option, int value, PAPI_option_t *ptr)`

This function sets specific options of the **PAPI** library, its substrate, or specific EventSets. The `PAPI_option_t` structure represents a union of all the structures that can be arguments to the different options. In addition, there may exist machine specific options so please check the header file and documentation. This function has a C binding only.

The following options are defined:

`PAPI_SET_MPXRES`

This option sets the multiplexing interval to `value`. `value` represents the time in microseconds between successive updates of the counting hardware. Values less than 1 millisecond may be rounded to the nearest possible resolution. A value of 0 disables multiplexing completely. Multiplexing is turned off initally. `ptr` should be set to `NULL.`

`PAPI_DEBUG`

This option turns on internal error reporting so that when one of **PAPI** functions returns an error code, `PAPI_perror(code,NULL,0)` is called implicitly before the function returns. Here `value` is one of `PAPI_QUIET`, `PAPI_VERB_ECONT` or `PAPI_VERB_ESTOP.` By default, this option is set to `PAPI_QUIET`, do nothing and return the error code to the calling function.

`PAPI_SET_OVRFLO`

This option enables PAPI to asynchronously deliver notification that the counter for a hardware Event has exceeded a user defined threshold. Here `value` represents the EventSet in which to enable this option.

The `overflow` member of the `PAPI_option_t` union structure contains the following members:

```
void (*handler)(int signal, siginfo_t *si, void *ucontext, int
EventSet, int Event, int count);
```

This function is called much like a signal handler. This is a function that takes all of the signal handler arguments as defined in the system’s header files, plus the EventSet, Event causing the overflow, and the current value of the counter for that hardware Event.

```
int count;
```

This is the threshold after which the handler is called.

```
int signal;
```

This is the signal used to activate the handler. The signal chosen often has side effects on the kind of information available in the `siginfo_t` structure. Please check your system header file and man pages carefully.

`int flag;`

This value tells how the overflow mechanism is to be used. The following options may not be mutually exclusive depending on their semantics.

`PAPI_ONESHOT`

This flag indicates that the library should not reinstall the overflow handler when an overflow is generated. By default this feature is disabled and the handler is called upon every detected overflow.

`PAPI_RANDOMIZE`

This flag indicates that the library should randomly choose a new overflow interval with maximum value of `count` after the handler is called for the first time. By default this feature is disabled.

`int PAPI_get_opt(int option, int *value, PAPI_option_t *ptr)`

This function queries the status of tunable options in the PAPI library. `value` is an input/output parameter. The `ptr` structure is solely for output. Not all options fill the `PAPI_option_t` structure. This function has a C binding only.

The following options are defined:

`PAPI_GET_MPXRES`

This option returns the current multiplexing interval in microseconds in `value`. An interval of 0 means that multiplexing is not enabled.

`PAPI_GET_OVRFLO`

This option fills the `overflow` member of the `PAPI_option_t` union structure as defined above. `value` is an input parameter defined as the EventSet to query.

The High Level API

The simple interface implemented by the following three routines allows the user to access and count specific hardware events from both C and Fortran. It should be noted that this API can be used *in conjunction with the low level API.* However, even if counter multiplexing is enabled by the user, the high level API is still only able to access those events countable simultaneously by the underlying hardware.

`int PAPI_start_counters(int *events, int array_len)`

Start counting the events named in the `events` array. This function implicitly stops and initializes any counters running as the result of a previous call to `PAPI_start_counters().` It is the user's responsibility to choose events that can be counted simultaneously by reading the vendor's documentation.

`int PAPI_read_counters(long long *values, int array_len)`

Read the running counters into the `values` array. This call implicitly initializes the internal counters to zero and lets them continue to run upon return.

`int PAPI_stop_counters(long long *values, int array_len)`

Stop the running counters and copy the counts into the `values` array.

Examples

Example 1: Pseudo code for a program that measures the cycles, floating point instructions and L1 data cache misses for the whole program and one segment of code.

```
#include "papi.h"

main()
{
    int EventSet = PAPI_NULL;
    long long first[2] = {0, 0}, total[2] = {0, 0};

    PAPI_add_event(&EventSet, PAPI_L1_DCM);
    PAPI_add_event(&EventSet, PAPI_FP_INS);
    PAPI_add_event(&EventSet, PAPI_TOT_CYC);

    PAPI_start(EventSet);

    /* Do something that’s really slow; */

    PAPI_accum(EventSet, first);

    /* Do something else even slower; */

    PAPI_stop(EventSet, total);
    PAPI_cleanup(&EventSet);
}
```

Example 2: Pseudo code for a program that generates a histogram of where overflows occur for L1 cache misses.

```
#include <signal.h>
#include "papi.h"
```

```
void handler(int signal, siginfo_t *si, void *ucontext, int EventSet, int Event, int
count)
{
    /* Get text address from si->xxxx;
    Get thread context from ucontext;
    Hash text address for our context into hash bucket;
    Add count for Events to our hash bucket; */

    return;
}

main()
{
    int L1_Cache_Misses = PAPI_NULL;
    PAPI_option_t options = { 0, };
    long long total[2] = { 0, 0 };

    PAPI_add_event(&L1_Cache_Misses, PAPI_L1_DCM);
    PAPI_add_event(&L1_Cache_Misses, PAPI_L1_ICM);

    options.overflow.handler = handler;
    options.overflow.count = 10001;
    options.oveflow.signal = SIGPROF;
    options.oveflow.flag = 0x0;

    PAPI_set_opt(PAPI_SET_OVRFLO, L1_Cache_Misses, &options);

    PAPI_start(EventSet);

    /* do_something; */

    PAPI_stop(EventSet, total);
    PAPI_cleanup(EventSet);

    /* decode hash table’s addresses into source lines;
    print histogram of lines vs. overflows; */

    exit(0);
}
```

Implementation Status

Reference implementations are currently underway for the SGI MIPS R10000, IBM Power, and Linux platforms, and are expected to be completed by April 1999. For current information see the PAPI home page.